MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

InnoChain: a Distributed Ledger for Industry with Formal Verification on all Implementation Levels

V. A. Kukharenko[1], K. V. Ziborov[2], R. F. Sadykov[2,3], A. V. Naumchev[3], R. M. Rezin[3], L. A. Merkin-Janson[3]

DOI: 10.18255/1818-1015-2020-4-454-471

[1]Moscow Institute of Physics and Technology, 9 Institutskiy per., Dolgoprudny, Moscow Region 141701, Russia.
[2]Lomonosov Moscow State University, 1 Leninskie Gory, Moscow 119991, Russia.
[3]Innopolis University, 1 Universitetskaya, Innopolis 420500, Russia.

MSC2020: 93A30, 68Q60
Research article
Full text in Russian

Received November 18, 2020
After revision December 2, 2020
Accepted December 16, 2020

The extent of formal verification methods applied to industrial projects has always been limited. The proliferation of distributed ledger systems (DLS), also known as blockchain, is rapidly changing the situation. Since the main area of DLSs' application is the automation of financial transactions, the properties of predictability and reliability are critical for implementing such systems. The actual behavior of the DLS is determined by the chosen consensus protocol, which properties require strict specification and formal verification. Formal specification and verification of the consensus protocol is necessary but not sufficient. It is required to ensure that the software implementation of the DLS nodes complies with this protocol. The verified software implementation of the protocol must run on a fairly reliable operating system. The so-called "smart contracts", which are an important part of the applied implementations of specific business processes based on DLSs, must be verifiable as well. In this paper, we describe an ongoing industrial project that will result in a DLS verified at least at the four technological levels described above. We then share our experience with the formal specification and verification of HotStuff, a leader-based fault-tolerant protocol that ensures reaching distributed consensus in the presence of Byzantine processes.

Keywords: Byzantine fault tolerance; distributed consensus; blockchain; TLA+; verification; model checking

INFORMATION ABOUT THE AUTHORS

Vladimir Aleksandrovich Kukharenko correspondence author	orcid.org/0000-0001-7377-7548. E-mail: vladimir.a.kukharenko@gmail.com Bachelor student.
Kirill Viktorovich Ziborov correspondence author	orcid.org/0000-0002-5676-9105. E-mail: an49x00@gmail.com Bachelor student.
Rafael Faritovich Sadykov correspondence author	orcid.org/0000-0002-2792-2465. E-mail: sadykovrf@gmail.com Ph.D. student.
Alexandr Vladimirovich Naumchev	orcid.org/0000-0002-7683-0751. E-mail: a.naumchev@innopolis.ru Assistant professor, Ph.D.
Ruslan Maratovich Rezin	orcid.org/0000-0002-0604-2645. E-mail: r.rezin@innopolis.ru MSc.
Leonid Albertovich Merkin-Janson	orcid.org/0000-0002-8427-2200. E-mail: l.merkin@innopolis.ru Professor at Innopolis University, Doctor of Mathematics, Delft University of Technology, Dr.

Funding: Ministry of Digital Development, Communications and Mass Media of the Russian Federation and Russian Venture Company (Agreement No. 004/20 dd. 20.03.2020, IGK 0000000007119P190002).

For citation: V. A. Kukharenko, K. V. Ziborov, R. F. Sadykov, A. V. Naumchev, R. M. Rezin, and L. A. Merkin-Janson, "InnoChain: a Distributed Ledger for Industry with Formal Verification on all Implementation Levels", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 454-471, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

InnoChain: распределенный реестр для индустриального применения с формальной верификацией на всех уровнях реализации

В. А. Кухаренко[1], К. В. Зиборов[2], Р. Ф. Садыков[2,3], А. В. Наумчев[3], Р. М. Резин[3], Л. А. Меркин[3]

DOI: 10.18255/1818-1015-2020-4-454-471

[1]Московский физико-технический институт, Институтский пер., д. 9, г. Долгопрудный, Московская обл., 141701 Россия.
[2]Московский государственный университет им. М.В. Ломоносова, Ленинские горы, д. 1, г. Москва, 119991 Россия.
[3]Университет Иннополис, Университетская, д. 1, г. Иннополис , 420500 Россия.

УДК 519.7
Научная статья
Полный текст на русском языке

Получена 18 ноября 2020 г.
После доработки 2 декабря 2020 г.
Принята к публикации 16 декабря 2020 г.

Степень применения методов формальной верификации в индустриальных проектах всегда была ограничена. Распространение систем распределенного реестра (СРР), известных также как блокчейн, быстро меняет ситуацию. Поскольку основной областью применения СРР является автоматизация финансовых транзакций, свойства предсказуемости и надежности являются критическими при реализации таких систем. Реальное поведение СРР определяется выбранным протоколом консенсуса, свойства которого нуждаются в строгой спецификации и формальной верификации. Формальная спецификация и верификация протокола консенсуса необходима, но недостаточна. Необходимо удостовериться, что программная реализация узлов СРР соответствует данному протоколу. Верифицированная программная реализация протокола должна запускаться на достаточно надежной операционной системе. Так называемые “умные контракт”, которые являются важной частью прикладных реализаций конкретных бизнес-процессов на основе СРР, также должны быть верифицируемы.В данной работе мы описываем реализующийся в настоящее время индустриальный проект, результатом которого станет СРР, верифицированная по меньшей мере на четырех описанных выше технологических уровнях. Мы также описываем наш опыт формальной спецификации и верификации протокола HotStuff – отказоустойчивого протокола для гарантированного достижения консенсуса в присутствии византийских процессов и лидера.

Ключевые слова: устойчивость к византийским условиям; распределенный консенсус; блокчейн; TLA+; верификация; проверка моделей

ИНФОРМАЦИЯ ОБ АВТОРАХ

Владимир Александрович Кухаренко автор для корреспонденции	orcid.org/0000-0001-7377-7548. E-mail: vladimir.a.kukharenko@gmail.com студент бакалавриата.
Кирилл Викторович Зиборов автор для корреспонденции	orcid.org/0000-0002-5676-9105. E-mail: an49x00@gmail.com студент бакалавриата.
Рафаэль Фаритович Садыков автор для корреспонденции	orcid.org/0000-0002-2792-2465. E-mail: sadykovrf@gmail.com аспирант.
Александр Владимирович Наумчев	orcid.org/0000-0002-7683-0751. E-mail: a.naumchev@innopolis.ru доцент, кандидат наук.
Руслан Маратович Резин	orcid.org/0000-0002-0604-2645. E-mail: r.rezin@innopolis.ru магистр.
Леонид Альбертович Меркин	orcid.org/0000-0002-8427-2200. E-mail: l.merkin@innopolis.ru профессор.

Финансирование: Министерство цифрового развития, связи и массовых коммуникаций РФ и АО "Российская венчурная компания"(договор №004/20 от 20.03.2020, ИГК 0000000007119P190002).

Для цитирования: V. A. Kukharenko, K. V. Ziborov, R. F. Sadykov, A. V. Naumchev, R. M. Rezin, and L. A. Merkin-Janson, “InnoChain: a Distributed Ledger for Industry with Formal Verification on all Implementation Levels”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 454-471, 2020.

Введение

Целью данной работы является публикация опыта применения формальной спецификации и верификации в условиях реализации крупного индустриального проекта (далее – *Проекта*). Наблюдения, приведенные в статье, могут помочь другим исследователям в области формальных методов в выборе правильного подхода к спецификации и верификации разрабатываемых систем. Мы начинаем изложение результатов с сравнительного анализа протоколов консенсуса для систем распределенного реестра (СРР) с целью выбора такого из них, который удовлетворял бы требованиям задач, которые стоят перед нами в рамках Проекта. Основной задачей в данном случае является задача формализации и автоматизации процесса оказания услуг по заправке воздушных судов (ВС) с целью контроля своевременной оплаты и минимизации рисков, связанных с отказом одной из сторон от исполнения своих обязательств.

В обобщенном виде задачу заправки ВС можно рассматривать как регуляцию отношений между двумя видами участников: потребители и поставщики услуг. Также, без ограничения общности, можно утверждать, что каждый участник отношений управляет узлом СРР и участвует в изменениях состояния СРР. Под состоянием СРР можно понимать некоторый набор переменных, характеризующий процессы реального мира. Например, это может быть объем средств (баланс) участников или этап, на котором находится заправка некоторого ВС. Состояние СРР хранится каждым узлом и изменяется специальными командами – *транзакциями* – которые были инициированы узлами сети. Состояние СРР не может изменяться произвольным образом, что контролируется протоколом консенсуса. Не нарушая общности, можно сказать, что *протокол консенсуса* – это набор правил обмена сообщениями специального вида между узлами СРР с целью согласования списка транзакций для исполнения и перехода в новое состояние. Тем не менее, описание СРР зависит от специфики управляемого процесса. Чтобы не приходилось разрабатывать отдельную систему под каждую задачу, вводится понятие *смарт-контракта*. Под *смарт-контрактом* понимается некоторая программа, написанная на специальном языке программирования, и описывающая, каким образом изменяется состояние отдельно взятого процесса реального мира. Код смарт-контракта также хранится в качестве состояния СРР. Например, целесообразно иметь разные смарт-контракты для управления услугами по заправке ВС и хранению топлива. Таким образом, смарт-контракты позволяют упростить разработку высоконадежных систем для потенциально не связанных между собой задач реального мира, используя, тем не менее, общую инфраструктуру сети и общую реализацию базовых протоколов взаимодействия между узлами сети. Исходя из этой модели, отдельные транзакции содержат вызовы определенных функций смарт-контрактов для перехода в новое состояние СРР из текущего. Практически во всех существующих системах СРР код смарт-контракта не может быть изменен после его развертывания. Это гарантирует каждому участнику, что ни одна из сторон не может отказаться от обязательств, даже если является разработчиком смарт-контракта.

Важно отметить, что каждый участник отношений действует, в первую очередь, исходя из максимизации собственной выгоды – настолько, насколько это позволяет протокол консенсуса. Класс протоколов консенсуса, которые управляют отношениями подобного рода, принято называть устойчивым к «византийским» условиям (Byzantine Fault Tolerant, BFT). Другим классом протоколов, которые используются на практике, являются протоколы, устойчивые к выходу из строя определенного набора узлов (Crash Fault Tolerant, CFT). В качестве примера CFT протокола можно привести Raft [1]. CFT протоколы являются более производительными по сравнению с BFT протоколами, но не позволяют справиться со злонамеренным поведением узлов, и поэтому неприменимы в рамках рассматриваемой задачи.

Современные СРР также подразделяются на открытые и закрытые. Открытые СРР не накладывают ограничения на возможность подключения произвольного узла к сети, в то время как в закрытых СРР такие ограничения присутствуют. Как правило, в закрытых СРР узлы либо фиксируются при

развертывании сети, либо поддерживаются специальные транзакции, позволяющие существующим узлам добавлять новые. Как можно догадаться, открытые СРР значительно сложнее закрытых из-за более общей задачи, которая ими решается. Например, в открытых СРР возникает проблема, связанная с неконтролируемым порождением «бесполезных» транзакций узлами, которые сложно идентифицировать в реальном мире и невозможно отключить от СРР. Данная атака может в конечном счете привести к отказу в обслуживании для правильных узлов. Напротив, в закрытых СРР правильные узлы могут исключить узел из сети в случае вскрытия факта злонамеренного поведения. Более того, на участника закрытых СРР могут накладываться контрактные обязательства в реальном мире, прежде чем другой участник инициирует транзакцию по добавлению нового узла в сеть. Таким образом, основным требованием к протоколу консенсуса в закрытых СРР является обнаружение злонамеренного поведения узла с сохранением корректного состояния системы. Стоит заметить, однако, что скорость работы в закрытых СРР сильно зависит от числа узлов (в лучшем случае зависимость линейная), но гарантируется финальность принятого состояния – принятые и исполненные в рамках протокола консенсуса транзакции, которые привели систему в данное состояние, не могут быть отменены ни при каких обстоятельствах. В открытых СРР нет подобной зависимости от числа узлов, но в каждый момент времени финальность состояния можно вычислить лишь с некоторой вероятностью. Другими словами, принятая и исполненная в рамках протокола консенсуса транзакция может быть отменена с некоторой вероятностью, которая уменьшается с ростом количества транзакций принятых после нее. Примерами открытых СРР являются Bitcoin [2], Ethereum [3] и EOS [4], закрытых – Hyperledger Iroha [5] и Fabric [6]. Важно отметить, что современные протоколы консенсуса для открытых СРР содержат часть, реализующую протокол консенсуса для фиксированного количества участников, который также применим в закрытых СРР. Таким образом, на базе закрытой СРР возможна разработка открытой СРР.

Возвращаясь к задаче заправки ВС, кратко сформулированной в начале раздела, стоит заметить, что участники заправочной сети – это компании, сертифицированные на оказание услуг, связанных с топливом, а также владельцы ВС. Между ними можно распределить некоторое ограниченное множество узлов для доступа к заправочной сети, поскольку такие компании могут быть не заинтересованы в издержках на содержание выделенного узла. С другой стороны, количество узлов важно с точки зрения безопасности сети. Например, часть протокола DPOS (Delegated Proof of Stake) консенсуса СРР EOS [4], реализующая протокол BFT-консенсуса для фиксированного количества участников, требует порядка 21 узла, допуская, таким образом, что не более 7 из них могут управляться злоумышленниками. Также важно заметить, что возникает необходимость выделения некоторого привилегированного множество узлов, которые будут управлять процессом добавления или удаления новых узлов, что автоматически делает СРР закрытой. Можно заключить, таким образом, что достаточно выбрать подходящий протокол консенсуса для закрытой СРР. Это не означает, однако, что закрытая СРР не может эволюционировать в открытую: как было замечено ранее, новое состояние открытой СРР может определяться фиксированным числом участников, но этот набор должен периодически переизбираться из общего числа участников некоторой дополнительной логикой на основе прошлых состояний СРР.

В данной статье мы: рассматриваем наиболее популярные протоколы BFT-консенсуса и взаимосвязи между ними (раздел 1); приводим обоснование выбора протокола консенсуса HotStuff, а также технологий TLA+/TLC для его верификации (раздел 2); анализируем две независимые попытки спецификации и верификации протокола, одна из которых завершилась успешно, а другая натолкнулась на некоторые проблемы, причины которых мы разбираем (раздел 3). Анализ обеих попыток позволит будущим исследователям избежать аналогичных проблем, а также повторно использовать практики, которые привели к положительным результатам.

Итоговая TLA+ спецификация протокола HotStuff, формально верифицированная с помощью инструмента TLC на предмет соответствия типовым требованиям безопасности (safety), будет служить формальным техническим заданием для программной реализации узла распределенного реестра InnoChain. Для формальной верификации самой программной реализации мы планируем использовать инструментарий HOL4 [7]. Для этой цели TLA+ спецификация протокола HotStuff, вместе с программной реализацией узла, будет транслирована в набор теорем HOL4. Процесс верификации будет состоять в доказательстве полученных теорем с использованием операционной семантики целевого языка программирования, выраженной в виде аксиом HOL4. Программную реализацию узла планируется собирать с помощью формально верифицированного компилятора с последующим запуском на операционной системе, управляемой формально верифицированным микроядром seL4 [8].

1. Устойчивость к византийским условиям

Под *протоколом консенсуса* будем понимать распределенный алгоритм синхронизации состояния СРР между узлами. В данной работе будем рассматривать протоколы BFT-консенсуса, которые предполагают наличие не более чем f узлов (из $n \geq 3f + 1$ [9]) с произвольным злонамеренным поведением. Узлы, которые следуют протоколу консенсуса, будем называть *правильными*. Каждую попытку перехода из одного состояния СРР в другое будем называть *раундом*. Как уже было сказано в предыдущем разделе, состояние СРР изменяется в результате исполнения транзакций, отправленных пользователями. Таким образом, в рамках раунда протокола консенсуса узлам необходимо договориться о едином списке транзакций для выполнения.

Раунды протокола консенсуса для удобства описания подразделяют на фазы, в каждой из которых узлы обмениваются сообщениями с некоторой целью. Далее скажем, что протокол консенсуса является *частично синхронным*, если действует предположение, что время доставки сообщения не превышает заранее установленного значения Δ по истечении некоторого времени стабилизации сети (Global Stabilization Time, GST). В противном случае, будем называть протокол консенсуса *асинхронным*. В данной работе все рассматриваемые протоколы, за исключением HoneyBadgerBFT, являются частично синхронными.

Сложность протоколов консенсуса привязана к количеству операций создания или проверки электронно-цифровой подписи, используемой при обмене сообщениями, так как данная операция зачастую занимает даже больше времени, чем передача данных по сети.

Наконец, корректность протокола консенсуса определяется двумя классами свойств: *безопасности* (*safety*) и *живости* (*liveness*). Свойства безопасности гарантируют, что для любого момента времени t найдется такой момент времени $t' \geq t$, что все правильные узлы будут иметь одно и тоже состояние, которое будет являться результатом исполнения списка транзакций, согласованных в рамках протокола консенсуса. Свойства живости гарантируют, что для любого момента времени t найдется такой момент времени $t' \geq t$, что состояние сети изменится при наличии корректных транзакций от пользователя.

Дальнейший разбор наиболее популярных протоколов BFT-консенсуса строится на основе эволюции одного из первых реализованных на практике протоколов – PBFT (Practical Byzantine Fault Tolerant). Таким образом, демонстрируются решения задач, которые вставали перед научным сообществом после применения очередных улучшений в рамках актуального на тот момент протокола консенсуса. Также мы рассматриваем наиболее актуальный асинхронный протокол консенсуса – HoneyBadgerBFT – и проводим его сравнение с частично синхронными протоколами.

1.1. PBFT

Practical Byzantine Fault Tolerance (PBFT) [10] относится к классу частично синхронных BFT протоколов консенсуса. Данный протокол определяет поведение трех сущностей: пользователь, узел и лидер. Допустим, имеется перенумерованное множество узлов $N = \{1, ..., n\}$, где n ($n \geq 3f + 1$) – количество узлов сети, а f – максимальное количество злонамеренных узлов. Тогда лидер для раунда r определяется по правилу: $l = r \pmod n$. Под пользователем понимается произвольное устройство, которое может инициировать транзакции в СРР, используя интерфейс узла, но при этом может и не участвовать в протоколе консенсуса.

Предлагается следующий алгоритм действия пользователя:

- Формируется подписанная транзакция $\langle REQUEST, o, t, c \rangle_{\sigma_c}$, где o – данные операции, t – метка времени, c – идентификатор пользователя, σ_c – подпись пользователя.
- На основе отслеживаемого номера раунда вычисляется узел, который в данный момент является лидером. Далее, именно этому узлу направляется транзакция.
- Ожидается $f+1$ сообщений от узлов вида $\langle REPLY, r, t, c, i, rs \rangle_{\sigma_i}$, где r – номер текущего раунда, rs – результат исполнения транзакции.
- Если не удалось получить $f + 1$ результатов за разумное время, то рассылаем запрос первого шага всем узлам.

Раунд протокола консенсуса состоит из следующих фаз: *предподготовка* (*pre-prepare*), *подготовка* (*prepare*), *фиксация* (*commit*).

Без ограничения общности считаем, что в рамках раунда узлы формируют одну транзакцию пользователя для исполнения. Успешный раунд протокола консенсуса выглядит следующим образом:

- Лидер формирует и отправляет остальным узлам сообщение вида: $\langle \langle PRE\_PREPARE, r, s, d \rangle_{\sigma_l}, m \rangle$, где r – номер текущего раунда, s – порядковый номер, присвоенный транзакции пользователя, m – транзакция пользователя, d – хэш-значение транзакции.
- Остальные узлы, получив сообщение *PRE_PREPARE*, проверяют его корректность. В случае, если узел не принимал для текущих значений r и n сообщение с другим d, то он переходит в фазу *prepare* и рассылает всем узлам сообщение: $\langle PREPARE, r, s, d, i \rangle_{\sigma_i}$
- Получив сообщение *PREPARE*, узел проверяет, соответствуют ли значения r, s и d сообщению *PRE_PREPARE*. После получения $2f + 1$ корректных *PREPARE* сообщений от разных узлов, лидер отправляет сообщение $\langle COMMIT, r, s, d, i \rangle_{\sigma_i}$
- После получения $2f + 1$ корректных $COMMIT$ сообщений от разных узлов – при условии исполнения транзакций с меньшим порядковым номером – команда, заключенная в транзакции m, исполняется, и состояние СРР на узле i изменяется.
- После исполнения транзакции узел i отправляет пользователю результат в *REPLY* сообщении, как было отмечено ранее.

Наряду со сменой лидера, которая будет описана ниже, протокол гарантирует следующее важное свойство: транзакция m с порядковым номером s, исполненная одним узлом, обязательно будет исполнена с тем же порядковым номером $f + 1$ корректными узлами – возможно, после нескольких смен лидера.

В каждой фазе узел запускает таймер с интервалом Δ, отмеченном в определении частичной синхронности. Данный интервал линейно увеличивается в случае срабатывания таймера и уменьшается, если фаза заканчивается раньше. В случае срабатывания таймера, узлом инициируется процесс смены лидера.

Также, в качестве оптимизации, каждые k раундов каждым узлом запускается процесс сохранения стабильного состояния, которое используется для снижения объема хранимой информации и для синхронизации «отставших» узлов. Порядковый номер транзакции, соответствующий послед-

нему стабильному состоянию, рассылается остальным узлам в сообщении $\langle CHECKPOINT, s, d, i \rangle_{\sigma_i}$ и запоминается локально после получения $2f + 1$ корректного сообщения.

Процесс смены лидера инициируется узлом путем рассылки сообщения вида $\langle VIEW\_CHANGE, r + 1, s, C, P, i \rangle_{\sigma_i}$, где s – порядковый номер последнего стабильного состояния (совпадает с порядковым номером соответствующей транзакции), C – множество из $2f + 1$ подписей, доказывающих корректность соответствующего s состояния, P – множество, состоящее из множеств P_m, соответствующих транзакции m, для которой узлом получено $2f + 1$ *PREPARED* сообщение, и имеющей превышающий s порядковый номер.

После получения лидером $2f + 1$ корректных сообщений $VIEW\_CHANGE$, локальное стабильное состояние обновляется на новое с максимальным порядковым номером. Далее, лидер формирует и рассылает сообщение $\langle NEW\_VIEW, r + 1, V, O \rangle_{\sigma_l}$, где: V – множество, содержащее $2f + 1$ полученных лидером сообщений $VIEW\_CHANGE$; O – множество, содержащее *PRE_PREPARE* сообщения для каждой транзакции из P с порядковым номером, превышающим соответствующий текущему стабильному состоянию номер. Если множество транзакций, удовлетворяющих этому условию, пусто, то в качестве хэш-значения используется специальное значение d_{null}, сообщающее, что лидер сменился, но нет команд для исполнения. Каждый узел, в свою очередь, проверяет корректность построения множества O и других данных при получении NEW_VIEW и запускает описанный раннее процесс из двух фаз для каждого *PRE_PREPARE* сообщения.

Нетрудно видеть, что при благоприятном исходе раунда алгоритм достижения консенсуса имеет сложность $O(n^2)$ в силу того, что на каждом из n узлов каждая фаза требует проверки $O(n)$ подписей. В случае смены лидера, необходимо повторить фазы для порядка n транзакций (так как число f злонамеренных узлов линейно зависит от общего числа узлов), что приводит к сложности порядка $O(n^3)$.

1.2. HoneyBadgerBFT

BFT-протокол консенсуса HoneyBadgerBFT [11] относится к классу *асинхронных*, что означает отсутствие каких-либо предположений о задержках СРР по сравнению с протоколом, рассмотренном в предыдущем разделе. Главная идея протокола заключается в использовании распределенного алгоритма надежной доставки. Таким образом, узлам не нужно использовать таймер для смены состояния по истечении некоторого промежутка времени, так как сообщение, отправленное в сеть, рано или поздно будет доставлено.

Допустим, имеется пронумерованное множество узлов $N = \{1, ..., n\}$, где n ($n \geq 3f + 1$) – количество узлов сети, f – максимальное количество злонамеренных узлов. Перед началом работы сети генерируются необходимые для работы порогового шифрования [12] ключи: открытый ключ PK и n секретных SK_i ключей для каждого узла. Пороговое шифрование, как будет показано далее, используется для предотвращения цензуры транзакций. Предполагается, что на каждом узле имеется неограниченный буфер, который накапливает транзакции пользователей.

Абстрактное описание протокола консенсуса выглядит следующим образом:

- При наступлении раунда r, узел i случайным образом выбирает $\lfloor b/n \rfloor$ транзакций из первых b транзакций буфера. Обозначим эти транзакции *proposed*.
- Выбранный набор транзакций шифруется публичным ключом пороговой подписи: v_i = TPKE.Enc(PK, proposed).
- v_i передается на вход протоколу консенсуса для раунда r ($ACS[r]$), задача которого заключается в согласовании общего подмножества из предложенных узлами шифрованных наборов транзакций – $\{v_j\}_{j \in S}$, $S \subset N$.
- После получения $\{v_j\}$, инициируется процесс расшифровки $ACS[r]$ для раунда r.
- Для каждого $j \in S$ вычисляется e_j = TPKE.DecShare(SK_i, v_j).

- Рассылается сообщение вида $\texttt{DEC}(r, j, i, e_j)$.
- При получении $f + 1$ сообщений $\texttt{DEC}(r, j, k, e_{k,j})$, расшифровывается набор транзакций $y_j = \texttt{TPKE.Dec}(\texttt{PK}, (k, e_{k,j}))$.
- Общий набор транзакций для раунда r вычисляется как: $\texttt{block}_r = \texttt{sorted}(\bigcup_{j \in S} y_j)$.
- Обработанные транзакции удаляются из буфера.

Протокол консенсуса для получения общего подмножества транзакций для узла i использует протокол надежной передачи данных (RBC) [13] двоичного консенсуса (BA) [14] и имеет следующий вид:

- При получении узлом значения v_i, оно рассылается с помощью RBC.
- При получении значения v_j, оно передается протоколу бинарного согласования BA для определения включения данного набора транзакций в финальное множество.
- Как только $N - f$ результатов согласования получено для разных v_j, остальные результаты получают 0 в качестве значения.
- Завершается доставка всех v_j от RBC, и в качестве результата возвращается множество $\{v_j\}$ с результатом BA, равным 1.

Автор оригинальной работы установил, что при $b = \Omega(\lambda n^2 log(n))$ (λ is a security paramete) протокол консенсуса в худшем случае имеет сложность $O(b)$. Также можно заметить, что протокол наиболее эффективен в условиях избытка транзакций, так как в этом случае высока вероятность выбора каждым узлом разных наборов транзакций, что увеличивает пропускную способность сети.

1.3. SBFT

Протокол консенсуса Scalable Byzantine Fault Tolerance (SBFT) [15] разрабатывался с целью оптимизации PBFT. По этой причине в данном разделе мы не останавливаемся детально на алгоритме, а рассмотрим лишь предложенные авторами протокола изменения относительно PBFT.

Во-первых, стоит заметить, что в каждом раунде протокола PBFT каждый узел проверяет подпись сообщений от других узлов, что имеет сложность $O(n^2)$, где n – число узлов в сети. По этой причине авторы SBFT предложили ввести дополнительные 2 роли узлов в каждой фазе помимо лидера: C-коллектор и E-коллектор. Задачей C-коллектора является накопление сообщений определенного назначения от других узлов, пока их количество не достигнет необходимого порога, а также рассылка одного результирующего сообщения узлам, которое содержит единственную подпись, согласованную с подписями из полученных сообщений. Таким образом, C-коллектор и остальные узлы в совокупности проверяют только $O(n)$ сообщений в случае успешности раунда. Объединение нескольких подписей в одну достигается за счет использования так называемых пороговых подписей [16]. Главным недостатком данного подхода является то, что перед началом работы сети узлам необходимо совместно сгенерировать открытый ключ, который зависит от совокупности секретных ключей каждого узла. По аналогии с C-коллектором, задачей E-коллектора является сбор сообщений от узлов с результатом исполнения транзакции, объединение подписей и отправка одного сообщения пользователю. Таким образом, пользователю достаточно получить единственное корректное сообщения от E-коллектора для того, чтобы удостовериться, что транзакция успешно исполнена в CPP, вместо ожидания $f + 1$ сообщения от узлов, как в PBFT.

Второй важной оптимизацией является добавление «оптимистичного пути» раунда. Состоит она в том, что если C-коллектор до срабатывания таймера получает *PREPARED* сообщения от всех n узлов, то он сразу формирует $FULL\_COMMIT\_PROOF$ сообщение и рассылает его узлам. В случае корректности сообщения, узел сразу добавляет транзакцию в очередь на исполнение, минуя дополнительную фазу *precommit* протокола PBFT. В случае срабатывания таймера быстрого пути, протокол переходит в C двумя фазами (как PBFT, но с учетом оптимизации подписей), при этом C-коллектор становится лидером.

Далее, авторы SBFT предлагают использовать не один коллектор, а c C-коллекторов, где c намного меньше n. Это увеличивает вероятность прохождения раунда по быстрому пути даже в случае недоступности некоторых коллекторов. В результате получается, что число узлов в сети $n = 3f + c + 1$, где f – число злонамеренных узлов, а c – число узлов, которые могут быть недоступны, но не могут совершать других зловредных действий, противоречащих протоколу консенсуса.

С учетом оптимизации подписей можно заключить, что SBFT имеет сложность $O(n)$ в случае позитивного исхода раунда и $O(n^2)$ – в случае смены лидера.

1.4. Tendermint

Протокол консенсуса Tendermint [17] строится на тех же фазах, что и PBFT, но при этом в каждом раунде протокола консенсуса узлы договариваются не об одной транзакции, а о блоке, состоящем из набора транзакций. Авторы Tendermint предлагают оптимизацию, которая помогает избежать сложностей с повторением фаз протокола консенсуса для каждого неуспешного блока после смены лидера. Это достигается за счет специально разработанного протокола рассылки сообщений (Tendermint Gossip protocol) и машины состояний, которая хранит только последний блок для повторения протокола консенсуса в новом раунде в случае смены лидера. В отличие от SBFT, Tendermint позволяет оптимизировать второе из двух узких мест протокола PBFT: отправка сообщения от каждого узла всем остальным и повторение фазы протокола консенсуса для каждой неуспешной транзакции при получении необходимого количества *PREPARE* сообщений от узлов CPP.

Протокол рассылки сообщений Tendermint использует предположение о частичной синхронности сети и гарантирует, что если правильный узел отправил сообщение в момент времени t, то остальные правильные узлы получат его не позднее, чем через $max\{t, \mathtt{GST}\} + \Delta$, где GST – некоторый момент времени, когда сеть стабилизируется, а Δ – гарантированный интервал доставки сообщения в стабильной сети.

Авторы Tendermint предлагают вместо названий фаз PBFT *pre-prepare*, *prepare*, *commit* названия *proposal*, *prevote*, *precommit* для лучшего понимания задач каждой фазы; между фазами, тем не менее, можно провести взаимно однозначное соответствие. Также машина состояний Tendermint подразумевает хранение следующих значений для оптимизации процесса смены лидера: *validValue*, *validRound*, *lockedValue*, *lockedRound*. Значения *validValue* и *validRound* хранят значение последнего блока транзакций, для которого было получено $2f + 1$ сообщений *prevote*, а также номер соответствующего раунда. Эти значения сбрасываются после получения $2f + 1$ PRE_COMMIT сообщений, то есть после согласования блока в рамках протокола консенсуса, но до этого момента правильный лидер рассылает эти значения в своем $proposal$ сообщении. Значения *lockedValue* и *lockedRound* обновляются исключительно в фазе *precommit*, перед отправкой сообщения PRE_COMMIT, и необходимы для отслеживания того факта, что не будет отправлено $PREVOTE$ сообщение на *PROPOSAL*, блок транзакций которого отличен от *lockedValue*.

Таким образом, протокол рассылки сообщений Tendermint, в совокупности с машиной состояний, гарантирует тот факт, что блок транзакций *lockedValue* рано или поздно будет принят сетью, так как после конечного количества раундов найдется правильный лидер, который разошлет *PROPOSAL* с *validValue* и *validRound*, согласованный с *lockedValue* и *lockedRound*. Данное сообщение получат остальные правильные узлы ввиду гарантий протокола передачи данных.

В результате, сложность протокола консенсуса Tendermint оценивается как $O(n^2)$ в общем случае, где $n \geq 3f+1$ – количество узлов сети, f – максимальное число злонамеренных узлов. Стоит заметить, однако, что эту сложность можно улучшить до $O(n)$, если использовать подход с пороговыми подписями SBFT.

1.5. HotStuff

У протокола консенсуса Tendermint отсутствует свойство «оптимистичного отклика» (Optimistic responsiveness) ввиду синхронной задержки, вызванной необходимостью гарантировать хоть какой-то отклик: после достижения GST, выбранному правильному лидеру достаточно получить $2f + 1$ сообщений о смене лидера, содержащих *validValue* для формирования *PROPOSAL* сообщения с блоком транзакций, который рано или поздно будет принят сетью. Следующий сценарий показывает отсутствие описанной проблемы при отсутствии синхронной задержки между раундами. Допустим, один из узлов получит $2f + 1$ *prevote* сообщений и обновит как *validValue*, так и *lockedValue*, и станет заблокированным, в то время как остальные узлы сменят лидера. Новый лидер не узнает актуального *validValue* и предложит другой блок транзакций, который отвергнет заблокированный узел. Более того, в этом новом раунде ситуация может повториться, и заблокируется еще один узел.

Описанная проблема возникает из-за того, что переменные *validValue* и *lockedValue* обновляются в одной фазе раунда. Авторы протокола HotStuff [18] предложили добавить еще одну фазу для решения этой проблемы и обновлять эти переменные в разных фазах. Таким образом, если узел блокируется (устанавливается переменная *lockedValue*), то это также означает, что не менее $2f + 1$ узлов установили переменную `validValue`, и корректный лидер включит эти транзакции в предложенные для нового раунда. Таким образом, в HotStuff узлы голосуют в фазах *prepare*, *pre-commit*, *commit* и применяют результат в фазе *decide*. Также вместо переменных *validValue* и *lockedValue* используются *prepareQC* и *lockedQC*. Помимо этого, HotStuff предполагает использование упомянутой не раз пороговой криптографии, чем и гарантирует линейную сложность работы. Сбором подписей и генерацией единой подписи занимается лидер раунда.

Успешный раунд протокола консенсуса HotStuff описывается следующим образом:

- Лидер формирует и отправляет остальным узлам сообщение вида $\langle PREPARE, curProposal, highQC \rangle_{\sigma_l}$, где $curProposal$ – блок транзакций, предлагаемый для согласования в текущем раунде, $highQC$ – доказательство того, что блок транзакций соответствует самому актуальному *prepareQC* в сети.
- Остальные узлы, получив сообщение *PREPARE*, проверяют его корректность. Если переменная узла *lockedValue* не установлена или соответствует *curProposal* и *highQC*, то узел отправляет сообщение $\langle VOTE\_PREPARE, curProposal \rangle_{\sigma_i}$.
- Получив $2f + 1$ сообщений $VOTE\_PREPARE$, лидер формирует пороговую подпись, обновляет *prepareQC* на основе *curProposal* и голосов узлов, отправляет сообщение $\langle PRE\_COMMIT, prepareQC \rangle_{\sigma_i}$.
- Остальные узлы, получив сообщение PRE_COMMIT, проверяют его корректность и подпись. В случае успеха, узел отправляет сообщение $\langle VOTE\_PRE\_COMMIT, prepareQC \rangle_{\sigma_i}$ и обновляет *prepareQC*.
- Получив $2f + 1$ сообщений $VOTE\_PRE\_COMMIT$, лидер формирует пороговую подпись, устанавливает *lockedQC* равной *prepareQC* и отправляет сообщение $\langle COMMIT, prepareQC \rangle_{\sigma_i}$.
- Остальные узлы, получив сообщение $COMMIT$, проверяют его корректность и подпись. В случае успеха, узел отправляет сообщение $\langle VOTE\_COMMIT, prepareQC \rangle_{\sigma_i}$ и обновляет *lockedQC*.
- Получив $2f + 1$ сообщений $COMMIT$, лидер формирует пороговую подпись и отправляет сообщение $\langle DECIDE, lockedQC \rangle_{\sigma_l}$. Кроме того, соответствующий *lockedQC* блок попадает в очередь на исполнение.
- Остальные узлы, получив сообщение *DECIDE*, проверяют его корректность и подпись. В случае успеха, узел добавляет соответствующий блок в очередь на исполнение.

Далее, авторы HotStuff заметили, что получившие *PREPARE* сообщения узлы могут оптимистично предположить, что предложенный блок будет успешно принят, и параллельно инициировать следующий раунд. Таким образом, блоки, ожидающие принятия, становятся зависимы друг от друга и могут быть представлены в виде списка. Каждый последующий блок содержит подписи узлов для предыдущего блока. Таким образом, если после данного блока в список добавились еще три блока, то можно считать этот блок принятым, так как это событие эквивалентно прохождению трех фаз протокола HotStuff. Данный алгоритм получил название Chained HotStuff.

Table 1. Comparison of partially synchronous BFT-consensus protocols

Таблица 1. Сравнение частично синхронных протоколов BFT-консенсуса

Протокол	Правильный лидер	Смена лидера	Оптимист. отклик
PBFT [10]	$O(n^2)$	$O(n^3)$	Есть
SBFT [15]	$O(n)$	$O(n^2)$	Есть
Tendermint [17]	$O(n^2)$	$O(n^2)$	Нет
HotStuff [18]	$O(n)$	$O(n)$	Есть

В таблице 1 приведено заключительное сравнение наиболее популярных частично синхронных протоколов BFT-консенсуса, используемых на практике. Алгоритмическая сложность измеряется в количестве проверок электронно-цифровой подписи за раунд в зависимости от числа узлов в сети $n \geq 3f + 1$, где f – максимально возможное количество злонамеренных узлов. Как было показано в предыдущих разделах, первопроходец PBFT имеет довольно высокую алгоритмическую сложность и сложную логику смены лидера. Авторы SBFT предложили оптимизацию с помощью пороговых подписей, а также оптимизацию *оптимистичным путем* для стабильных сетей, но при этом сложность логики смены лидера осталась той же. Далее, авторы Tendermint предложили существенное упрощение логики смены лидера, но при этом жертвуют свойством *оптимистичного отклика* и вносят зависимость от протокола передачи данных (Tendermint gossip) для корректной работы. Наконец, протокол HotStuff предлагает решение проблем протокола Tendermint введением дополнительной фазы, но с сохранением линейной сложности.

Отдельно стоит отметить, что асинхронные протоколы на данный момент имеют более высокую алгоритмическую сложность ($\Omega(\lambda n^2 log(n))$ - HoneyBadgerBFT) и сложнее в реализации.

2. Верификация Распределенного Консенсуса

Проведенное в разделе 1 сравнение протоколов консенсуса говорит о том, что HotStuff является лучшим выбором в качестве протокола консенсуса для реализации закрытой системы распределенного реестра с точки зрения сложности работы и наличия свойства оптимистичного отклика. Стоит заметить, однако, что ввиду необходимости использования криптографического стандарта ГОСТ 34.10 и отсутствия в нем решения для пороговых подписей, реализация HotStuff планируется без них. С одной стороны, это увеличит алгоритмическую сложность до $O(n^2)$, с другой – упростит формальную верификацию реализации протокола. Далее заметим, что все рассмотренные протоколы BFT-консенсуса с линейной сложностью предполагают использование пороговых подписей, но либо не имеют свойства оптимистичного отклика (как Tendermint), либо имеют более высокую сложность для фазы смены лидера (PBFT, SBFT) в сравнении с Hotstuff.

Помимо сложности работы протокола консенсуса и времени его реализации, необходимо оценить его формальную верифицируемость: насколько сложно доказать, что программный код алгоритма имеет свойства, требуемые его формальным описанием – спецификацией. В качестве примера можно рассмотреть не раз упомянутое свойство безопасности. Тогда одной из задач формальной верификации протокола консенсуса является математическое доказательство того, что распреде-

ленная система никогда не попадет в такую ситуацию, когда два разных правильных узла сети будут иметь противоречащие друг другу состояния (например, один узел будет сообщать, что заправка ВС закончилась успешно, а другой – что не успешно). На сегодняшний день существует два подхода к формальной верификации протоколов консенсуса: верификация методом проверки моделей (model checking [19] и верификация дедуктивными методами [7, 20, 21]).

Метод проверки моделей применяется не к исходному коду протокола, который используется для компиляции и исполнения, а к абстрактной модели протокола в виде структуры Крипке. Каждому состоянию автомата соответствует набор свойств протокола, которые верны в данном состоянии. Доказательство того, что построенная модель имеет заданное свойство, осуществляется путем представления этого свойства в виде формулы в одной из темпоральных логик – например, LTL (логика линейного времени) или CTL (логика ветвящегося времени). Для каждой из логик существует отдельный алгоритм, который проверяет свойство для заданной структуры Крипке. Результатом работы данного алгоритма является либо сообщение о том, что свойство выполняется, либо возвращение последовательности переходов (контрпример) в автомате, которая демонстрирует нарушение свойства. Преимуществом подхода проверки моделей является то, что структура Крипке представляет собой более формальную запись протокола консенсуса, нежели некоторый псевдокод, который используется в научных статьях и спецификациях для объяснения идеи работы протокола, что позволяет находить недостатки либо в момент формирования модели, либо при исследовании контрпримера после завершения процесса проверки модели. Одним из недостатков данного подхода является то, что зачастую модели протоколов консенсуса в первоначальной записи имеют огромное число состояний, что не позволяет завершить алгоритм верификации за приемлемое время, в результате чего возникает задача упрощения исходной модели. Другим недостатком метода является то, что проверке подвергается не исходный код протокола, а его абстрактная модель, в результате чего возникает задача доказательства соответствия проверенной модели ее программной реализации.

Напротив, дедуктивные методы верификации чаще всего применяются к программной реализации, но требуют разработки операционной семантики языка реализации в выбранной системе верификации. Операционная семантика необходима для получения информации о том, каким образом та или иная синтаксическая конструкция языка программирования изменяет состояние программы. Дедуктивный подход к верификации использует идею традиционного формирования доказательства математическими методами, но с возможностью автоматизации шаблонных техник – тактик (например, метод математической индукции), которые являются составной частью *доказателей* (proof assistant). Таким образом, формальная верификация дедуктивным методом подразумевает написание программы для протокола консенсуса на некотором языке программирования и перенос этого кода в доказатель, в котором уже описана операционная семантика для данного языка, с последующей попыткой автоматического доказательства с помощью встроенных в доказатель тактик. Главным недостатком данного подхода в сравнении с проверкой моделей является то, что процесс не подразумевает автоматического нахождения контрпримера – то есть либо удается доказать нужное свойство для программы, либо не получается; при этом неудачная попытка не означает, что такого доказательства не существует.

На момент написания данной работы нам удалось обнаружить результаты верификации протоколов Tendermint и PBFT. В работе по верификации Tendermint авторы используют метод проверки моделей для доказательства того, что построенная модель протокола в системе TLC имеет свойство безопасности. Напротив, для верификации PBFT применялся метод дедуктивной верификации в системе Coq. Более того, был разработан специальный фреймворк, Velisarius [22], который можно применять для верификации произвольного BFT протокола консенсуса. Velisarius представляет собой набор теорий для доказателя Coq [21], в которые встроены основные абстракции протоколов

консенсуса: например, теория *EventOrdering* для описания событий, происходящих на разных узлах сети, или теория *Process*, в которой определены машины состояний узлов. С помощью Velisarius было доказано свойство безопасности PBFT, и из полученных спецификаций сгенерирован исполняемый код протокола на языке OCaml.

Несмотря на имеющиеся результаты формальной верификации Tendermint и PBFT, мы приняли решение использовать Hotstuff в качестве основного протокола системы распределенного реестра, реализуемой в рамках Проекта, ввиду наличия свойства оптимистичного отклика и упрощенной реализации фазы смены лидера. Стоит заметить, что схожесть Tendermint и Hotstuff позволяет применить идеи, использованные при проверке модели протокола Tendermint, для верификации модели протокола Hotstuff. Помимо применения проверки моделей, необходимо также использовать дедуктивные методы для верификации исходного кода протокола – например, используя фреймворк Velisarius.

3. Спецификация и Верификация Протокола HotStuff с Помощью TLA+/TLC

Так как протоколы HotStuff и Tendermint очень похожи, то первая версия модели была реализована на основе работы Игоря Коннова [23]. Ввиду того, что она фокусировалась на свойстве безопасности, то с этого и началась работа. Все последующие рассуждения, как и проверка модели в целом, основываются на том, что мы доверяем представлению реальной модели поведения в TLA+. Мы детально описываем сделанные нами допущения и модификации, а также обосновываем наличие таких допущений. Мы не включаем код наших TLA+ спецификации в статью для экономии пространства. Обе спецификации находятся в открытом доступе на GitHub [24].

3.1. Попытка 1

В модели, описываемой в этом разделе [25], была предпринята попытка добавления древовидной структуры команд в явном виде. Согласно статье, описывающей протокол HotStuff [18], каждая команда является полем некоторой структуры данных – вершины. Вершина так же содержит множество хэш-значений всех предыдущих команд, таким образом определяя отношение частичного порядка на множестве вершин. Будем говорить, что вершина B *актуальнее* вершины A, если множество хэш-значений из A является подмножеством хэш-значений из B. При построении описываемой модели криптография HotStuff была опущена (в пороговых подписях опущено поле sig) для достижения разумного времени проверки модели. С той же целью хэш-значения предшествующих команд были заменены на множество самих команд.

При таком подходе к построению вершины она будет содержать множество всех команд, прошедших фазу *PREPARE*, что сильно скажется на числе состояний модели и, следовательно, увеличит время проверки модели протокола. Однако, согласно первоначальному замыслу, данный подход может упростить спецификацию свойств протокола при ещё одном допущении: команды добавляются в множество вместе с их порядковым номером. Нужный номер команде назначает лидер в фазе *PREPARE*, а влияние византийского лидера будет обнаружено корректными узлами с помощью предикатов *ExtendsFrom* и *SafeNode*. Таким образом, при выбранном подходе вершина состоит из родительской ссылки – множества пар команд и соответствующих им порядковых номеров – и выбранной в текущем раунде команды. Данная оптимизация заметно сокращает множество возможных значений вершины, а значит и множество возможных состояний модели при верификации. Также стоит заметить, что оптимизация не мешает перенести свойство частичного порядка определенное ранее: вершина B является *актуальнее* вершины A, если упорядоченная по номеру последовательность команд из A является префиксом упорядоченной по номеру последовательности команд из B.

3.1.1. Предикаты SafeNode и ExtendsFrom

Узел в фазе *PREPARE* ожидает сообщения типа *PREPARE* от лидера, а затем проверяет полученную вершину с помощью предикатов *ExtendsFrom* и *SafeNode*.

Двуместный предикат *ExtendsFrom* в описываемой модели выполняет проверку следующего свойства вершины.

- Новая вершина актуальнее вершины из предыдущего раунда;
- Если к множеству пар команд вершины из предыдущего раунда добавить ее выбранную команду с соответствующим номером, то полученное множество совпадет с множеством пар команд новой вершины;

В случае, если указанные свойства выполнены, *ExtendsFrom* обращается в *TRUE*.

Предикат *SafeNode* реализован аналогично псевдокоду HotSuff [18]: он получает на вход вершину, проверяет ее актуальность и соответствие раунда вершины текущему раунду узла.

3.1.2. Спецификация свойств

Для спецификации свойств протокола, по аналогии с Tendermint, была введена переменная *decision*. Для каждого процесса *decision* хранит множество пар команд, которые будут исполнены машиной состояний узла, и множество соответствующих им порядковых номеров. Изменение переменной *decision* происходит только в фазе *DECIDE* узла после получения от лидера сигнала на исполнение команд (сообщения типа *DECIDE* с *CommitQC*). Это изменение заключается в добавлении в *decision* пары, состоящей из принятой на исполнение команды и соответствующего ей номера.

Описанная модель позволяет легко специфицировать свойство корректности (agreement), заключающееся в том, что все корректно работающие узлы получают одинаковое значение после исполнения принятой команды. Другими словами у любых двух корректных узлов их множества *decision* пусты (состояние не изменилось), либо между ними можно построить биекцию.

3.1.3. Итоги

В результате запуска проверки описываемой модели в TLC была выявлена ошибка, связанная с неправильной реализацией отправки всевозможных сообщений с новой пороговой подписью византийскими узлами. Кроме того, стало очевидно, что такой подход повлечёт за собой генерацию большого количества состояний, а проверка будет занимать длительное время. По этой причине необходимо исследовать варианты оптимизации полученной модели HotStuff.

3.2. Попытка 2

Первая проблема, с которой пришлось столкнуться при разработке альтернативной спецификации [26] – использование пороговых подписей в HotStuff. Она привела к необходимости различать подписанные и неподписанные сообщения, а также наложила ряд ограничений на византийские узлы – пришлось сделать допущение о невозможности подделать поле с пороговой подписью. Это не позволяет добавлять все сообщения византийских узлов в начале работы алгоритма. Решением данной проблемы является возможность добавлять эти сообщения по мере продвижения алгоритма, то есть был добавлен переход в машине состояний, при котором, если это возможно, все византийские узлы отправляют всевозможные сообщения с новой пороговой подписью.

Вторая проблема – наличие древовидной структуры предложенных команд (используется в предикате *ExtendsFrom*), которая формируется при обработке новых команд лидером. Попытка добавления этой структуры в явном виде описана в предыдущем подразделе. Такой подход многократно увеличивал число состояний, а также усложнял описание работы византийских узлов. Поэтому эта проблема была решена позже, и временно проверялась лишь корректность команды.

3.2.1. Проблемы с оптимизацией

После решения первой проблемы, написанная спецификация почти полностью соответствовала модели поведения. Были добавлены служебные состояния и функции, но сути это не меняло. Появилась другая проблема – проверка этой модели в TLC с византийскими узлами занимала слишком много времени (даже при $N = 4$). Это был ожидаемый результат, так как по сравнению с Tendermint увеличилось число фаз, а также усложнилась подделка сообщений византийскими узлами. Все это создавало слишком много новых состояний.

В качестве решения предлагается упрощение модели и использование следующего допущения (на основании оригинального алгоритма [18]):

- Когда ожидаются $N - F$ голосов от узлов, нам важен лишь факт получения сообщения, а не личность отправителя – поэтому был введен специальный массив-счетчик (*msgsCount*). Кроме того, это сильно упрощает подделку пороговых подписей для византийских узлов, так как они формируются аналогично. Важно отметить, что мы не допускаем копий и везде учитываем лишь первую отправку сообщения.
- В некоторых фазах – $PRECOMMIT$, $COMMIT$ и $DECIDE$ – лидер не делает ничего полезного, кроме создания пороговых подписей для доказательства получения сообщений от узлов, поэтому в этих фазах было необходимо отказаться от позиции лидера. Теперь все узлы ожидают нужного количества сообщений от других узлов, а не пороговой подписи от лидера (и реализуется это через *msgsCount*).
- После предыдущих упрощений понятно, что поле фазы (в оригинальной работе $type$) в пороговой подписи становится не используемым и по этой причине можно от него избавиться.
- Также видно, что в фазе *PREPARE* лидером никак не используется поле *node*. Это поле также было ликвидировано.

Кроме того, был немного переработан способ хранения отправленных сообщений для фазы *PREPARE*. Получились массивы, которые в полях отправленных сообщений хранили множество прикрепленных пороговых подписей (функция: отправитель -> раунд -> вершина листа с командой -> множество полей $justify$).

В результате время работы TLC значительно сократилось, и стал виден прогресс исследования состояний модели. Хотя в определении *ExtendsFrom* все еще оставалась заглушка в виде проверки корректности отправленной команды, удалось запустить модель на различных значениях и найти ошибку в TLA+ спецификации. Также удалось проверить корректность на 4 узлах (за сутки работы на сервере, что является хорошим результатом для продолжительности верификации подобных моделей) и получить ошибки при различных N и большом количестве византийских узлов (больших F).

3.2.2. Добавление древовидной структуры команд

Для целей оптимизации количества состояний модели было использовано предположение, что ветвление больше двух делать нецелесообразно ввиду того, что если удается получить ошибку при большем ветвлении, то ошибка должна обнаружиться и без него. Также было использовано предположение что количество ветвлений должно быть конечным.

Для реализации этого замысла двоичное дерево задается на этапе создания модели. В результате можно не создавать новые команды динамически, что позволяет избежать новых состояний, а также упрощает написание предиката $ExtendsFrom$. Также была исключена проверка корректности предложенных команд, так как она слишком сильно зависит от конкретной реализации протокола.

Теперь все команды, предлагаемые для согласования в рамках протокола консенсуса, берутся из двоичного дерева. В такой ситуации, когда все команды корректные, ошибка может произойти лишь из-за принятия неверной ветви команд одной или несколькими вершинами. Из-за этого

изменились проверяемые инварианты. Свойство $Validity$ теперь говорит, что все принятые узлом вершины продолжают друг друга. Свойство “Agreement” теперь говорит, что все узлы приняли вершины из одной ветви. Теперь спецификация учитывает все необходимые нам детали протокола.

Древовидная структура добавила много новых состояний, из-за чего время работы (при $N = 4$, на сервере) увеличилось приблизительно до трех недель. Но это разумное время, которое позволяет проверить необходимые случаи при небольших значениях N.

3.2.3. Возможное развитие модели

Нами была рассмотрена возможность спецификации и верификации свойства живости. Для проверки этого свойства необходима модель с конечным числом состояний, что привело к двум вариантам:

1. Обобщить протокол, чтобы сократить роль раундов – пока не совсем ясно, как это делать; кроме того, это будет серьезным отступлением от оригинальной модели поведения.
2. Ограничить число раундов, как и в случае свойства безопасности. Это понизит качество проверяемого условия, но позволит проверить при конкретных значениях. Для этого варианта нужно не так много изменений в спецификации для свойства безопасности, что делает его предпочтительным.

Таким образом, для проверки свойства живости предлагается изменить инвариант в спецификации. Предполагается, что он должен гласить следующее: если все узлы оказались в одном раунде, лидер корректный, а таймер NEW_VIEW не успевает сработать (можем такое предполагать, так как считаем, что GST наступило, а также прошло необходимое время для увеличения таймера), то решение будет принято.

4. Заключение

В данной статье мы:

- Привели краткое описание индустриального проекта по построению распределенного реестра, обязательной частью которого является формальная верификация всех основных компонентов разрабатываемой системы.
- Проанализировали популярные протоколы распределенного консенсуса, применимые к нашей задаче.
- Провели обзор применяемых на практике методов формальной верификации таких протоколов.
- Объяснили выбор протокола HotStuff и метода проверки моделей для его верификации.
- Описали и проанализировали две независимые попытки спецификации HotStuff в TLA+ с последующей проверкой выполнения требуемых свойств в TLC (инструментарий TLA+/TLC является стандартом де-факто в проверке моделей распределенных протоколов).

Мы надеемся, что проведенный нами анализ сэкономит время и силы других исследователей, занимающихся реализацией похожих проектов.

References

[1] M. Fazlali, S. M. Eftekhar, M. M. Dehshibi, H. T. Malazi, and M. Nosrati, «Raft Consensus Algorithm: an Effective Substitute for Paxos in High Throughput P2P-based Systems», *CoRR*, vol. abs/1911.01231, 2019.

[2] S. Nakamoto, «Bitcoin: A peer-to-peer electronic cash system», Manubot, Tech. Rep., 2019.

[3] G. Wood *et al.*, «Ethereum: A secure decentralised generalised transaction ledger», *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] E. Elrom, «EOS.IO Wallets and Smart Contracts», in *The Blockchain Developer*, Springer, 2019, pp. 213–256.

[5] F. Muratov, A. Lebedev, N. Iushkevich, B. Nasrulin, and M. Takemiya, «YAC: BFT Consensus Algorithm for Blockchain», *CoRR*, vol. abs/1809.00554, 2018.

[6] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, *et al.*, «Hyperledger fabric: a distributed operating system for permissioned blockchains», in *EuroSys*, ACM, 2018, 30:1–30:15.

[7] T. Gauthier, C. Kaliszyk, and J. Urban, «TacticToe: Learning to Reason with HOL4 Tactics», in *LPAR*, ser. EPiC Series in Computing, vol. 46, EasyChair, 2017, pp. 125–143.

[8] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, «seL4: formal verification of an OS kernel», in *SOSP*, ACM, 2009, pp. 207–220.

[9] L. Lamport, R. Shostak, and M. Pease, «The Byzantine Generals Problem», *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[10] M. Castro and B. Liskov, «Practical Byzantine Fault Tolerance», in *OSDI*, USENIX Association, 1999, pp. 173–186.

[11] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, «The honey badger of BFT protocols», in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 31–42.

[12] J. Baek and Y. Zheng, «Simple and efficient threshold cryptosystem from the Gap Diffie-Hellman group», in *GLOBECOM*, IEEE, 2003, pp. 1491–1495.

[13] M. Ben-Or, B. Kelmer, and T. Rabin, «Asynchronous secure computations with optimal resilience», in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, ACM, 1994, pp. 183–192.

[14] A. Mostefaoui, M. Hamouna, and M. Raynal, «Signature-Free Asynchronous Byzantine Consensus with $t < n/3$ and $O(n^2)$ Messages», in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, ACM, 2014, pp. 2–9.

[15] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, «SBFT: a Scalable Decentralized Trust Infrastructure for Blockchains», *CoRR*, vol. abs/1804.01626, 2018.

[16] D. Boneh, B. Lynn, and H. Shacham, «Short signatures from the Weil pairing», *Journal of cryptology*, vol. 17, no. 4, pp. 297–319, 2004.

[17] E. Buchman, J. Kwon, and Z. Milosevic, «The latest gossip on BFT consensus», *CoRR*, vol. abs/1807.04938, 2018.

[18] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, «Hotstuff: Bft consensus with linearity and responsiveness», in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[19] Y. G. Karpov, *MODEL CHECKING. Verification of parallel and distributed software systems*, 2010.

[20] L. C. Paulson, *Isabelle: A generic theorem prover*. Springer Science & Business Media, 1994, vol. 828.

[21] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, *et al.*, «The Coq proof assistant reference manual», *INRIA, version*, vol. 6, no. 11, 1999.

[22] V. Rahli, I. Vukotic, M. Völp, and P. Esteves-Verissimo, «Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq», in *ESOP*, ser. Lecture Notes in Computer Science, vol. 10801, Springer, 2018, pp. 619–650.

[23] I. Konnov, *Model Checking Tendermint*, 2020. [Online]. Available: https://github.com/informalsystems/verification/tree/igor/fork/spec/fork-cases.

[24] V. Kukharenko, K. Ziborov, and R. Rezin, *HotStuff TLA+ Specifications*. [Online]. Available: https://github.com/RZRussel/hotstuff-model-checking.

[25] K. Ziborov, *HotStuff TLA+ Specifications*, 2020. [Online]. Available: https://github.com/RZRussel/hotstuff-model-checking/blob/master/HotStuffAlpha.tla.

[26] V. Kukharenko, *HotStuff TLA+ Specifications*, 2020. [Online]. Available: https://github.com/RZRussel/hotstuff-model-checking/blob/master/ApalacheHotStuffCutted.tla.